Continuous Evolution of Multi-tenant SaaS Applications: A Customizable Dynamic Adaptation Approach

Fatih Gey, Dimitri Van Landuyt, and Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Heverlee, Belgium
firstname.lastname@cs.kuleuven.be

Viviane Jonckers
Software Languages Lab, Vrije Universiteit Brussel,
B-1050 Brussels, Belgium
ve.joncke@soft.vub.ac.be

***Abstract*—Applying application-level multi-tenancy in Software-as-a-Service (SaaS) offerings yields a number of compelling benefits: sharing a single instance of the application between large numbers of customer organizations increases cost efficiency and allows the SaaS provider to attain true economies-of-scale benefits. There is however a main downside to this: increased sharing of resources causes the SaaS application to be very difficult to modify after initial development and deployment without affecting service continuity: any change potentially affects the service levels promised to all enrolled tenant organizations and their end users.**

This rigidity is a key impediment as now the SaaS provider must evolve and maintain the SaaS offering at run time, on a gradual, per-tenant basis. This in turn causes a reality of multiple co-existing versions of individual components and as such introduces substantial management complexity.

To address these challenges, this paper motivates and defines key requirements that allows per-tenant, SLA-aware and gradual upgrades in the context of multi-tenant SaaS applications. In addition, we define an approach that allows the involved stakeholders (tenants, SaaS operators, SaaS developers, etc.) to customize the dynamic enactment of upgrades, and provide a number of alternative software upgrade strategies that represent different service quality trade-offs.

I. Introduction

In cloud computing, Software-as-a-Service (SaaS) applications are often designed to be multi-tenant, meaning that a single run-time application instance is shared among many customer organizations, the so-called *tenants*. The main benefit is that it allows the SaaS provider to optimize resource utilization, increase cost efficiency and, as such, leverages the ensuing economies-of-scale benefits to approximate the cloud computing ideal of commodity computing in which the cost of computing resources and infrastructure can be removed from the equation entirely.

The flip side of this coin, however, is that this typically leads to one-size-fits-all SaaS offerings that –once successful– have become very difficult to change without causing service disruptions for the subscribed tenant organizations and their end users. In many multi-tenant SaaS applications, the SLA violation and service disruption caused by updating and upgrading the entire SaaS application for the sake of maintenance and evolution is unacceptable. For example, waiting for application-wide quiescence [1] is practically infeasible in a successful SaaS offering that optimizes resource utilization and therefore is continuously servicing multiple tenants at once.

This highlights the need for powerful run-time adaptation mechanisms that allow the SaaS provider to update and upgrade the SaaS offering in a gradual manner, on a fine-grained, per-tenant basis. As opposed to a more classical discrete software update scheme (in which the application is brought from a consistent initial state to a consistent final state after updating in a one-shot operation), this scheme of continuous evolution causes the multi-tenant SaaS application to be permanently in an intermediate state of update. In this reality, multiple versions of the same components co-exist, older versions still servicing ongoing tenant requests, while new versions already service other tenants. Orchestrating this type of continuous update scheme involves dealing with ongoing tenant requests, under the constraints of specific tenant requirements, tenant SLAs and internal component dependencies (with the purpose of eventually phasing out older versions of the software) is a non-trivial task that involves substantial management complexity for the SaaS provider.

In presence of this complexity, different stakeholders of a given upgrade (e.g. tenant and SaaS operator) must rely on self-management capabilities [2] in order to understand and influence the impact of the upgrade. As unforeseen upgrades may have an impact on the application's effective *service quality*, the ability to trade-off different qualities is highly desired when established SLAs cannot be guaranteed during upgrade. As a result, the upgrade enactment mechanisms must additionally be customizable on a per-upgrade-case basis by different stakeholders. Current SaaS middleware [3] lacks support for this.

In this paper, we first motivate and define key requirements for gradual SLA-aware evolution of a long-running multi-tenant SaaS application. Then, we define a dynamic adaptation approach that addresses these requirements and deals with the complexity of multiple co-existing versions of components in order to leverage maximal service continuity throughout evolution for each tenant. Next, we provide support for different software upgrade strategies of which each represents a

different service quality trade-off.

This paper is structured as follows: in Section II, we discuss the background concepts related to Multi-tenant Software-as-a-Service, software evolution, and tenant SLAs. In Section III, we identify four key of requirements and formulate the problem statement for this paper. Section IV presents our customizable approach for gradual dynamic upgrades, and sections V and VI discuss related work and conclude the paper.

II. Background

We have studied a number of SaaS applications in the context of collaborative research projects with industry [4]–[6]. One example is a document processing service [4], [7], a B2B SaaS application that generates, distributes, signs, and archives different kinds of documents (e.g. paychecks, invoices) on behalf of tenant organizations.

Another example is a log-management-as-a-service application [5] that provides log storage and analysis services to tenant organizations. Logs are generated from distributed sources and provided in high volume to the SaaS application, which then analyses and archives them. The tenant can choose to be notified about urgent events (e.g. potential security incidents), or can access the SaaS provider's analysis application to obtain information on the health and behaviour of the monitored systems.

Based on these experiences, we observed that multi-tenant SaaS applications are commonly built as service-oriented architectures (SOA) without central coordination [8]. The supporting SOA middleware commonly offers (i) a service repository which stores service implementations, (ii) a service registry which maintains a view on running instance in that SOA, and (iii) to support multi-tenancy, a tenant configuration repository.

The approach presented in this paper is strongly aligned to this view on multi-tenant SaaS applications [9, Chapter 3]. In this setting, a tenant's request to the application typically initiates a chain of internal services. The set of operations performed by (multiple) services in order to process a single tenant request is referred to as a *tenant transaction*[1]. Further, the first time a service is being invoked for a certain transaction (called the *initial service call*) is distinguished from the subsequent times (*non-initial service calls*).

Evolving Multi-tenant SaaS applications: Software that is used over an extended period of time faces evolution scenarios of different types: *updates* that patch the application's lack of, for example, security or correctness are the most common, and replace only the implementation without altering its interface contracts or logic. Over time, however, changing (business) requirements must be addressed [12]. Such an *upgrade* has typically a higher level of invasiveness, and can be subdivided into those that preserve previous application abstractions while only adding features (called forward-compatible [13]), and those that do not. In the remainder of the paper, we refer to the latter simply as *upgrades* and to the former as *simple upgrades*.

[1]Note that the use of the term '*transaction*' here refers to its common use in the context of dynamic updates [1], [10], [11], as opposed to the contexts of ACID properties in databases, and transactions in distributed systems.

A. Impact on Service Quality

In many cases, a dynamic upgrade will impact the service quality of the application under upgrade [14]. Long-running, highly available services, such as multi-tenant SaaS applications, which will eventually encounter challenging upgrades [12], therefore have to consider quality degradation that results in a temporary violation of the SLA between provider and tenants.

Service Continuity: In this paper, we focus on *high-availability* and *version-consistent behaviour* [11] (i.e. consistent behaviour specified by one application version, as a form of correctness) as service qualities that are agreed up-on (SLA) between a SaaS provider and its tenants. To maintain both qualities (and therefore comply with SLAs) throughout an upgrade, the multi-tenant SaaS application (a) must ensure on-going tenant requests and (b) must keep accepting and processing new requests. From a tenant perspective, requirements (a) and (b) which aim to opposite goals, describe the same *observed* characteristic: availability or – when in context of long-running services – service continuity. Fox and Brewer [15] introduced a notion of availability in presence of failures which applies to our SaaS context as follows: *Yield* is the percentage of requests that are processed, while *Harvest* represents the completeness per request. In cases for which maximum yield *and* harvest is not feasible [16], *less yield may be traded-in for more harvest or vice versa* [15].

Time Coupling: Tenants may have different requirements about acceptable time windows for (i) maintenance in which the service may be not or less available, and for (ii) adapting their business processes and software to the changes.

Application of SLA vs. Nature of Upgrades: In practice, established SLAs for a SaaS application differ per tenant – a fact that remains unaccounted in one-shot upgrades when facing the challenge to guarantee this (these) very SLA(s). Given the nature of an upgrade, the strict application of SLAs may be infeasible (e.g. invasive upgrades), unnecessary (e.g. too costly while not avoiding any harm), or even undesired (e.g. a tenant or SaaS provider may be willing to sacrifice certain service qualities if that avoids bigger harm to the one's or the other's business).

III. Problem Statement

Once deployed and successful, a multi-tenant SaaS application is a long-running service that must adhere to high-availability among other quality requirements. Because a single application instance is shared among many tenants, application-wide dynamic upgrades in one shot are no longer feasible, as they impact the service to all tenants simultaneously in the same way. For example, waiting for application-wide quiescence [1] simultaneously for all tenants will involve unacceptable service disruption, cause violation of many tenant SLAs and eventually incur loss of revenue (and business).

This leaves as only alternative option a gradual evolution approach in which the enactment of upgrades is performed on a tenant-per-tenant basis and can be customized for each upgrade by involved stakeholders of the SaaS application. For such an approach, we identify the following four requirements of which three are optimization problems. After each requirement is defined, their interaction is discussed.

(R1) Tenant Upgrade Isolation: A tenant-by-tenant gradual evolution approach must isolate tenant scopes from each other such that while *one or more tenants T_u of the SaaS application may experience change*, for all *remaining tenants T_r it must be maintained unchanged*.

(R2) Support for Service Continuity: A dynamic evolution of a multi-tenant SaaS applications with minimal impact on its service continuity is challenging when the set of permitted upgrades is not limited. Especially, upgrades of services that add new features cause a reality of co-existing versions. In such cases, we require a mechanism that allows *making different trade-offs between yield and harvest during upgrade*.

(R3) Stakeholder Control: Different roles are involved in the upgrade process: SaaS developers should be able to control the deployment mechanism of the upgrade. SaaS operators should be able to enforce critical upgrades to be mandatory for enactment. Tenants should be able to align the time and the potential impact of an upgrade to their business context. In short, the upgrade support should be *adaptable to different kinds of upgrades* and sufficiently *flexible and customizable to enable an expected behaviour of the application throughout the upgrade process*.

(R4) High Automation: The economies-of-scale effects of multi-tenant SaaS applications motivate putting more effort in development in order to reduce operation costs [17]. Therefore, a *highly automated upgrade enactment mechanism* that is guided by meta-information of the services-under-upgrade is *preferred over a dynamic upgrade strategy that treats these components as a black box*.

Requirement Interactions: The capability of a service to accept preferences from the stakeholders themselves (known as *self service* is common for multi-tenant SaaS applications [18], and does not only provide Stakeholder Control (R3) but also also High Automation (R4). However, R4 and R3 are opposed goals: While automation decreases manual effort and temporal interdependency of upgrade and stakeholders, stakeholder control increases them. Therefore, a trade-off must be made between both. Service continuity (R2) is an optimization problem that may benefit from different stakeholders input (R3), e.g. while SaaS provider and tenant are able to provide high-level goals for the upgrade enactment, the SaaS developer may provide parameters for its low-level implementation.

Multi-tenant SaaS applications are built on two fundamentally different types of middleware: Self-hosted solutions [19], [20], or PaaS offerings [3]. None of these families of middleware provide explicit support for customizable upgrade mechanisms at tenant level. In this paper, we present a dynamic and gradual upgrade approach for multi-tenant SaaS applications that addresses these four key requirements. Tenant-aware

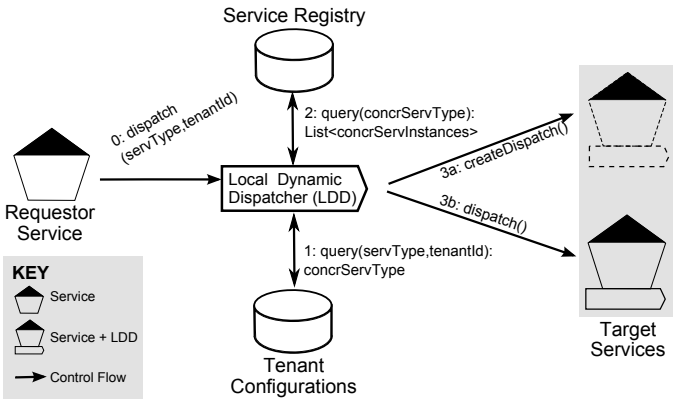

Fig. 1: Dynamically Dispatching a Local Service's Call.

dynamic service composition is a core enabling mechanism with which we define alternative upgrade strategies in the next section.

IV. Customizable Gradual Dynamic Upgrades

In this section, first, we introduce our tenant-aware dynamic composition mechanism on which we rely to present a set of strategies for gradual upgrades. Finally, we shortly present our initial middleware architecture in which we implemented these strategies, and discuss our approach in relation to the requirements we enumerated in Section III.

A. Tenant-aware Dynamic Composition

Our on-demand composition of services at run time works as follows: each service has a *Local Dynamic Dispatcher* component, which is triggered to forward a call to a desired service, as shown in Figure 1. Upon a forward request, the *requestor service*'s dispatcher takes a *service type* and a *tenant identifier* as input and performs the followings steps:

1. Resolve the given service type to a *concrete service type and version* using the *tenant configuration*.
2. Query the service registry for available *instances of the concrete service type/version*

3a. If no instances are available, trigger their launch

3b. Otherwise proceed with one of the available services

Stateful Services and Sticky Replication Instances: The aforementioned dispatch to a service instance assumes a stateless service; a schema in which it is of no relevance to which specific replica of a service instance a request is dispatched. However, sticky instances, i.e. a service requestor's requests are always dispatched to the same instance, can be easily supported by introducing session UIDs that are stored by the dispatcher in step 3a, and used in step 3b for the successor request. For simplicity, this feature is not discussed in the remainder of this paper.

B. Upgrade Strategies

This section presents a set of upgrade strategies to support different nuances of service continuity (trade-offs between yield and harvest) for a set of upgrade types (R2). The

illustrative presentation of the strategies focusses on a single upgrade at which the *current* service version β depicts the services before the upgrade and the *new* version δ those after the upgrade. At the time at which the upgrade starts, tenant transactions that have been at least partially processed by services of the current version are referred to as *in-progress transactions*.

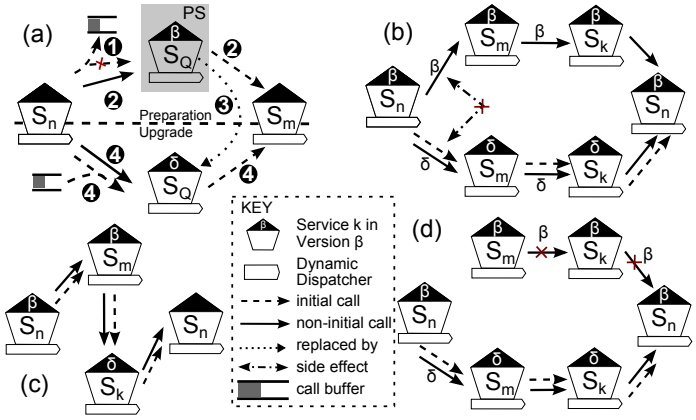

Fig. 2: Strategies for Different Upgrade Types.

(S1) Passivate and Queue (aka Impose Quiescence): This strategy implements the quiescence [1] type of dynamic upgrades in two-phases, as illustrated in Figure 2a: In the preparation phase, the set of services that are affected by the upgrade, which is called the passivation set PS[2], are isolated from processing incoming requests. Specifically, initial calls on services of PS are queued but not dispatched (1) in order to converge PS to an idle state, called *quiescence*. Non-initial calls, however, *are* dispatched to PS, while the system is waiting for in-progress transactions to complete[3] (2). In the update phase, $S_Q \in PS$ is first replaced by version δ (3), and queued calls are processed along with other calls (4).

As this strategy does not introduce new service versions before in-progress transactions are completed, it is the most safe of presented strategies not relying on any assumption of compatibility between current and new service versions. Faults caused by unanticipated side-effects of interactions between current and new service versions (aka mixed-version deployment [22]) are made impossible, which provides a maximum harvest. On the downside, it imposes the lowest yield during the upgrade: while the system is approaching quiescence, new transactions are not processed which renders the service effectively unavailable.

(S2) ¬Passivate and Process in Parallel with both Versions: For (groups of) transactions that cause no (incompatible) side-effects towards each other, current and new service versions could be deployed in parallel (see Figure 2b). For example, this is trivially true for stateless services, as shown by Li et al. [14]. In such a case, waiting for the application to "flush out" transactions that are being processed by the current version is no longer necessary. However, in order to converge to the new version, initial calls (i.e. new transactions) must be processed only by the new version. This strategy scores high in terms of yield and, if the condition above is true, also in terms of harvest.

(S3) ¬Passivate and Upgrade Transactions: In case current and new service versions are sufficiently compatible (e.g. when neither interfaces nor state semantics change), in-progress transactions may be dispatched to new service versions, as shown in Fig. 2c, without affecting yield or harvest of the application. An example are *updates* as defined in Sec. II. Applications with co-existing component versions that interact with each other and do not hold the condition above have already been studied [16], [22], its limits being pointed out by Ajmani et al. [16].

(S4) Discard In-progress Transactions: For urgent upgrades (e.g. security related) an extreme strategy is considered: at the time of upgrade, all in-progress transactions are discarded (cf. Fig. 2d). As a result, shortly after the start of the upgrade, only transactions using new service versions can be found in the system. This strategy imposes a yield of zero for a short amount of time, and comparably low average yield and harvest during the upgrade in favour of the upgrade urgency.

C. Middleware Architecture

A gradual dynamic upgrade on a per-tenant basis requires decoupling the new service version deployment (which affects all tenants as they share the same application deployment) and its use by specific tenants. The latter is called *activation* and is mainly driven by a tenant configuration which specifies service versions selected by the tenant.

Typically, such a tenant configuration is based on a constraint model (provided by the SaaS developer) which sets available service versions in relation such that only compatible service versions lead to valid configurations.

Our proposed middleware to support upgrade activation for multi-tenant SaaS application under the requirements R1–R4 is shown in Figure 3. In the following illustration, we assume that the new version β of the service implementation has already been deployed (i.e. stored in the service repository). We illustrate next the 4 steps of an upgrade activation, one by one.

(1) At first, a tenant administrator (or the SaaS provider operator) creates a new tenant configuration including new service versions. (2) Then, this configuration is compared to the previous configuration of that tenant, and the impact is determined relying on constraint models of both versions. (3) The tenant administrator (or the SaaS operator respectively) can review this impact in order to decide for or against the activation, for an activation time and to select the strategy of

[2]Originally [21], PS is defined as $PS = Q \cup NQ$, where Q is the component to be replaced and NQ its neighbours that may engage in transactions with Q. Passivation of PS aims at avoiding invocations from NQ to Q. However, passivation of NQ is not strictly necessary to avoid said invocations [10]; we provide passivation by blocking selected calls in the dispatcher component, and, hence, define $PS = Q$.

[3]In SOA applications, a service invocation is typically time-bound, i.e. it is completed when either the service returns a result or fails to do so within a time-out period.

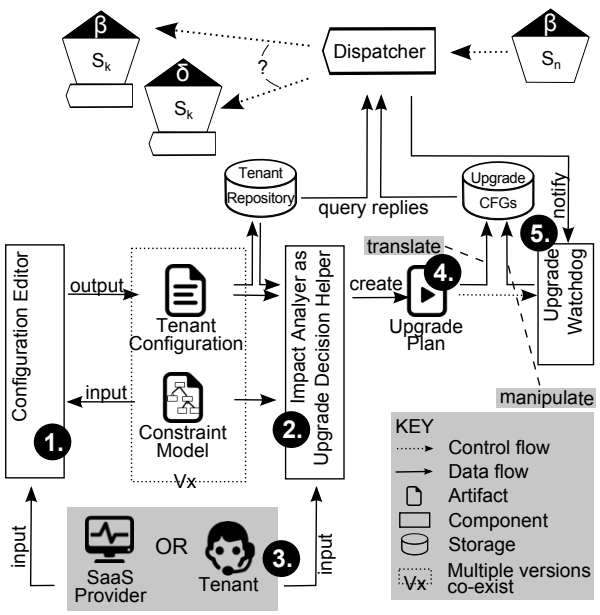

Fig. 3: Middleware for Gradual Upgrades.

that upgrade. In case the impact of that upgrade is unacceptable, another option at that point is to discard the upgrade activation, and to review it at another point in time or to skip this upgrade entirely. The latter requires the upgrade to be optional; otherwise (e.g. SaaS operator has flagged it as mandatory) the middleware would fall back to a default option (e.g. denying service) when the upgrade is not activated in time. For those cases in which the upgrade activation should proceed, an upgrade plan stores the chosen time and strategy. (4) The upgrade plan is then translated into a set of recomposition directives for the local dispatcher components and stored in a upgrade configuration database. (5) The *Upgrade Watchdog* (UW) which is notified about the upgrade plan, is concerned with the dynamism of the recomposition directives. For example, strategy S1 defines two phases whose transition is dependent on the application's state. In such a case, the UW is responsible for observing the application state (by reflecting on the dispatcher's notifications) and, when the transition is due, to deactivate the directives of the first phase and activate those of the second. In addition, it must remove all obsolete directives, once the upgrade can be considered complete so that directives of one upgrade do not interfere with those of upcoming ones.

Illustration of Reconfiguration Directives: Table I shows an example set of recomposition directives for each strategy (first column). Columns 2, 3, and 4 specify the dispatch request origin to which an directive applies: The value S_{im}^{tn} in column 2 (*service*) indicates that the directive applies only to specific services' dispatchers – typically these are the services that interact with the services under upgrade –, whereas an asterisk ($*$) specifies application to all dispatchers. Column 3 (*version*) refers to the current tenant configuration version of the transaction that triggered the dispatch request. Column

Strategy	Dispatch Request Origin			Method	Target Version
	Service	Version	Call Type		
$S1_{prep}$	S_{im}^{tn}	current	non-initial	dispatch	current
	S_{im}^{tn}	current	initial	queue	–
$S1_{upg}$	S_{im}^{tn}	current	(non-)initial	dispatch	new
$S2$	S_{im}^{tn}	current	initial	dispatch	new
	S_{im}^{tn}	current	non-initial	dispatch	current
	S_{im}^{tn}	new	non-initial	dispatch	new
$S3$	$*$	current	(non-)initial	dispatch	new
$S4$	S_{im}^{tn}	current	initial	dispatch	new
	$*$	current	non-initial	discard	–

TABLE I: Example Recomposition Directives.

4 (*call type*) limits the application of the given directive to the role of the dispatch request within a transaction: initial or non-initial service call? A directive can also apply to any of the two roles. Column 5 (*method*) dictates the method to be used by the dispatcher; if the value is 'dispatch', than the tenant configuration of the version specified in column 6 (*target version*) must be consulted in order to resolve a concrete service type and version. Otherwise, the dispatcher must 'queue' a service call (and actively reevaluate for finally dispatching it), or 'discard' it.

Note that, using this mechanism, (unlike our illustrative example) multiple services (even structurally interdependent ones [23]) could be upgraded at once.

D. Discussion

Our approach addresses requirements R1-R4 (Section III) for SLA-aware and gradual evolution of a multi-tenant SaaS application with different concepts: Via the mechanism of tenant configurations, tenants on the same SaaS application deployment can be serviced with different service versions (R1) and thereby upgraded independently (in time and structure) from other tenants.

This key capability enables different stakeholders to optimize upgrade enactment to individual (groups of) tenants without imposing drawbacks to others. The flexibility to apply a certain upgrade strategy for one tenant while using another strategy for a different tenant is an example of an optimization that aims at maximal service continuity (R2). The concrete benefits of an upgrade strategy strongly depends on the nature of the upgrade itself. Therefore, the role of the SaaS developer, who has the best understanding of the SaaS application, is in charge of providing alternative upgrade strategies that score well in service continuity (R2, R3) and are reusable for all tenants (R4).

Finally, our middleware design facilitates making a cost-effective trade-off between stakeholder control and high automation: the application complexity is processed up to a points at which user decisions become necessary. Different stakeholder are able to provide their decisions (R3): the SaaS developer incorporates his decision prior to the upgrade enactment by (not) implementing alternative upgrade strategies; the SaaS operator may flag an upgrade as mandatory and specify its latest activation time, while the tenant and/or SaaS

operator are able to ultimately schedule or reject the activation during the enactment. The dynamic (re)composition technique in our middleware applies the scheduled and fine-grained recomposition directives autonomously (R4).

V. Related Work

Dynamic Software Update research has a long history: while early works aimed at avoiding a shut down to update an application [13], [24], other approaches focus on state consistency [1] and minimal impact on high-availability [10], [11], [23], [25]. Supporting both characteristics (as well as different trade-offs between them) is challenging. Typically, formal approaches score well in both and specifically target update safety [25]–[27]. They are, on the other hand, limited to specific types of updates (e.g. forward-compatible ones) and usually involve memory-invasive operations impacting modularity and applicability on, for example, PaaS platforms.

Being a long-running service with high-availability requirements, multi-tenant SaaS applications face differing types of upgrades while aiming at maximal service continuity. Our approach therefore offer the SaaS provider a number of alternative strategies of which one can be selected, depending on the context (nature of the update, tenant requirements, SLAs, etc).

Service Variability [28] has been exploited to adapt a SOA [8] at run time, be it to maintain SLAs by switching to other third party services [29], or to reconfigure (non-)functional behaviour [30], most recently in the context of Dynamic Software Product Lines [31], [32]. None of these techniques deal with unanticipated changes that are common evolving to new requirements. In addition to the support of these evolution scenarios, our approach could be used as an enabler for service variability.

Self-managed systems represent a subset of *dynamic adaptation* approaches that address the complexity of a large and interconnected system in an automated way [21] so that in addition to monitoring, aggregation and analysis of relevant data for a decision on an adaptation plan, the decision itself is determined automatically (MAPE-K loop [2]). Such approaches rely on models that guide that decision. These models are either static [31]–[33], may evolve at run time [34], or are run-time artifacts that are dynamically constructed from a high-level specification of (business) goals [35]. Other approaches apply control theory [36] to determine a decision based on an a-priori-defined intent. In contrast, the SLA-aware evolution approach of multi-tenant SaaS applications described in this paper requires adaptation parameters to be based on the current run-time business context of the stakeholders in addition to the agreed service quality level that typically does not anticipate the application under upgrade. As such, a fully automated self-management approach is not applicable.

Evolution in the context of *Software-as-a-Service* or *Cloud Applications* has been approached from different perspectives: Update safety has been identified as challenge in the presence of (a) dynamically typed languages which are common in cloud applications (such as JavaScript) [26] and (b) co-existing components of different application versions [16]. Focussing on update-safety, the former is applicable only to a certain types of updates, while the latter [16] decreases application functionality in favour of update safety. Our concept targets any type of upgrades and provide rather fine-grained degradation trade-off capabilities. Dumitras et al. [22], [27] identify the scale of enterprise cloud applications as too large for one-shot upgrades, and highlight a set of challenges for consistency when upgrades are performed incrementally (*rolling upgrades* [37]) that lead to *mixed-version* systems [22]. While they propose to move the entire application to a "parallel universe" to avoid inconsistencies, we define a service in a SOA application as (smallest) evolution unit and provide a concept to deal with multiple co-existing versions. Others [32], [33] describe an adaptation and evolution of a SaaS application such that the change has been anticipated for the application, which effectively represents applicability to a subset of upgrade types.

VI. Conclusion and Future Work

Evolving a long-running SaaS application for all involved tenants is challenging and yet a necessity to support the service continuity essential to the SaaS business model. In this paper, we refine this challenge into a set of four key requirements, which we address by supporting dynamic software upgrades on a gradual, per-tenant basis. Further, we highlight that different types of upgrades are differently challenging in presence of those requirements, and we present a set of complementary upgrade strategies which are illustrated further.

Our approach provides strong benefits for both tenants and SaaS providers: individual tenants are given the flexibility to influence the upgrade process in terms of timing and strategy, while dealing with individual tenant requirements w.r.t. software evolution is (i) maximally automated and (ii) maximally offered as self-service [18], which both contribute to cost-effectiveness for SaaS providers. As such, the presented approach enables continuous evolution of multi-tenant SaaS applications and provides more flexibility to SaaS providers in reacting to changing requirements.

Future work: This paper fits into our ongoing research on the topic of service-oriented product lines for multi-tenant SaaS applications, called Service Lines [17]. In future work, we will further explore how the dynamic composition mechanism described in this paper can also serve as a customization technique to accommodate tenant-specific requirements at run-time [38], complete our on-going prototype efforts, and evaluate in a realistic case.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, the ADDIS research program funded by KU Leuven GOA, D-BASE and AIRCO. The D-BASE project is co-funded by iMinds (Interdisciplinary Institute for Technology), and AIRCO is supported by the Fund for Scientific Research-Flanders (FWO).

REFERENCES

[1] J. Kramer and J. Magee, "The evolving philosophers problem: dynamic change management," *Software Engineering*, 1990.

[2] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[3] S. Walraven, E. Truyen, and W. Joosen, "Comparing paas offerings in light of saas development," *Computing*, vol. 96, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00607-013-0346-9

[4] iMinds, "iMinds CUSTOMSS Project, http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=CUSTOMSS," Jan. 2013.

[5] iMinds, "DMS2 Project: Decentralized Data Management and Migration of SaaS, http://www.iminds.be/en/projects/2014/03/06/dms2," Website, 2014. [Online]. Available: http://www.iminds.be/en/projects/2014/05/06/d-base

[6] ——, "D-BASE Project: Optimization of Business Process Outsourcing Services, http://www.iminds.be/en/projects/2014/05/06/d-base," Website, 2014. [Online]. Available: http://www.iminds.be/en/projects/2014/05/06/d-base

[7] F. Gey, S. Walraven, D. Van Landuyt, and W. Joosen, "Building a Customizable Business-Process-as-a-Service Application with Current State-of-Practice," in *Software Composition*, 2013. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/394196

[8] H. Gomaa and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures," in *Software Architecture, WICSA*, June 2004, pp. 79–88.

[9] S. Walraven, "Middleware and methods for customizable saas (middleware en methodes voor aanpasbare saas)," Ph.D. dissertation, June 2014. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/455179

[10] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *Software Engineering*, Dec. 2007.

[11] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent dynamic reconfiguration of component-based distributed systems," in *FOSE*. ACM, 2011.

[12] M. Lehman, J. Ramil, P. D. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Software Metrics Symposium*, Nov 1997.

[13] T. Bloom and M. Day, "Reconfiguration and module replacement in argus: theory and practice," *Software Engineering Journal*, vol. 8, no. 2, pp. 102–108, 1993.

[14] W. Li, "Evaluating the impacts of dynamic reconfiguration on the qos of running systems," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2123 – 2138, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121211001439

[15] A. Fox and E. Brewer, "Harvest, yield, and scalable tolerant systems," in *Hot Topics in Operating Systems*, 1999.

[16] S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *ECOOP*. Springer, 2006. [Online]. Available: http://dx.doi.org/10.1007/11785477_26

[17] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen, "Efficient customization of multi-tenant software-as-a-service applications with service lines," *Journal of Systems and Software*, vol. 91, pp. 48–62, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121214000326

[18] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives," in *Proceedings Services-2*. IEEE, 2008.

[19] "OpenShift, https://www.openshift.com," Jan. 2015.

[20] "CloudFoundry, http://www.cloudfoundry.org," Jan. 2015.

[21] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 259–268.

[22] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system," in *Middleware*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1656980.1657005

[23] E. Truyen, N. Janssens, F. Sanen, and W. Joosen, "Support for distributed adaptations in aspect-oriented middleware," in *AOSD*, 2008. [Online]. Available: http://doi.acm.org/10.1145/1353482.1353497

[24] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *Software Engineering*, 1996.

[25] E. B. Johnsen, M. Kyas, and I. C. Yu, "Dynamic classes: Modular asynchronous evolution of distributed concurrent objects," in *Formal Methods*. Springer, 2009.

[26] P. Bhattacharya and I. Neamtiu, "Dynamic updates for web and cloud applications," in *APLWACA '10*. ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1810139.1810143

[27] I. Neamtiu and T. Dumitras, "Cloud software upgrades: Challenges and opportunities," in *MESOCA'11*, Sept 2011.

[28] T. Nguyen, A. Colman, M. A. Talib, and J. Han, "Managing service variability: state of the art and open issues," in *VaMoS'11*. ACM, 2011.

[29] K. Geebelen, S. Michiels, and W. Joosen, "Dynamic reconfiguration using template based web service composition," in *MW4SOC '08*. ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1462802.1462811

[30] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1077

[31] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, 2008.

[32] I. Kumara, J. Han, A. Colman, and M. Kapuruge, "Runtime evolution of service-based multi-tenant saas applications," in *Proceedings ICSOC*. Springer, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45005-1_14

[33] J. García-Galán, L. Pasquale, P. Trinidad, and A. Ruiz-Cortés, "User-centric adaptation of multi-tenant services: Preference-based analysis for service reconfiguration," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 2014. New York, NY, USA: ACM, 2014, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/2593929.2593930

[34] F. Chauvel, N. Ferry, B. Morin, A. Rossini, and A. Solberg, "Models@runtime to support the iterative and continuous design of autonomic reasoners." in *Proceedings Models@Run.time*, 2013.

[35] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, "Models@run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, Oct 2009.

[36] M. Shaw, "Beyond objects: A software design paradigm based on process control," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 1, pp. 27–38, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/225907.225911

[37] E. Brewer, "Lessons from giant-scale services," *Internet Computing, IEEE*, vol. 5, no. 4, pp. 46–55, Jul 2001.

[38] F. Gey, D. Van Landuyt, S. Walraven, and W. Joosen, "Feature models at run time: Feature middleware for multi-tenant saas applications," in *Models@run.time*, 2014.